Contents

Installing Gateways

7 minute read page test

Along with creating a service mesh, Istio allows you to manage gateways, which are Envoy proxies running at the edge of the mesh, providing fine-grained control over traffic entering and leaving the mesh.

Some of Istio's built in configuration profiles deploy gateways during installation. For example, a call to

`istioctl install` with `default settings` will deploy an ingress gateway along with the control plane. Although fine for evaluation and simple use cases, this couples the gateway to the control plane, making management and upgrade more complicated. For production Istio deployments, it is highly recommended to decouple these to allow independent operation.

Follow this guide to separately deploy and manage one or more gateways in a production installation of Istio.

Prerequisites

This guide requires the Istio control plane to be installed before proceeding.

You can use the `minimal` profile, for example `istioctl install --set profile=minimal`, to prevent any gateways from being deployed during installation.

Deploying a gateway

Using the same mechanisms as Istio sidecar injection, the Envoy proxy configuration for gateways can similarly be auto-injected.

Using auto-injection for gateway deployments is recommended as it gives developers full control over the gateway deployment, while also simplifying operations. When a new upgrade is available, or a configuration has changed, gateway pods can be updated by simply restarting them. This makes the experience of operating a gateway deployment the

same as operating sidecars.

To support users with existing deployment tools, Istio provides a few different ways to deploy a gateway. Each method will produce the same result. Choose the method you are most familiar with.

As a security best practice, it is recommended to deploy the gateway in a different namespace from the control plane.

IstioOperator | Helm | Kubernetes YAML

First, setup an `IstioOperator` configuration file, called `ingress.yaml` here:

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: ingress
spec:
  profile: empty # Do not install CRDs or the control pla
ne
  components:
    ingressGateways:
    - name: ingressgateway
      namespace: istio-ingress
      enabled: true
      label:
        # Set a unique label for the gateway. This is req
uired to ensure Gateways
```

```
      # can select this workload
      istio: ingressgateway
  values:
    gateways:
      istio-ingressgateway:
        # Enable gateway injection
        injectionTemplate: gateway
```

Then install using standard `istioctl` commands:

```
$ kubectl create namespace istio-ingress
$ istioctl install -f ingress.yaml
```

Managing gateways

The following describes how to manage gateways after installation. For more information on their usage, follow the Ingress and Egress tasks.

Gateway selectors

The labels on a gateway deployment's pods are used by `Gateway` configuration resources, so it's important that your `Gateway` selector matches these labels.

For example, in the above deployments, the `istio=ingressgateway` label is set on the gateway pods. To apply a `Gateway` to these deployments, you need to select the same label:

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: gateway
spec:
  selector:
    istio: ingressgateway
...
```

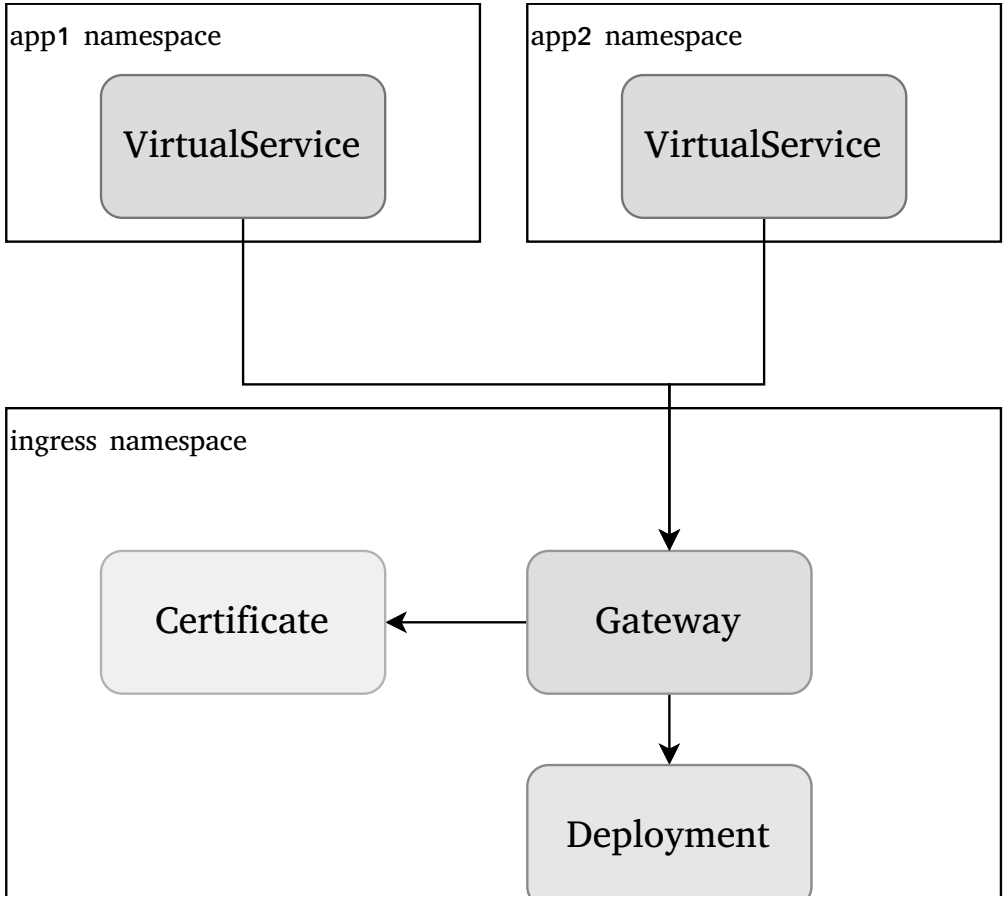
Gateway deployment topologies

Depending on your mesh configuration and use cases, you may wish to deploy gateways in different ways. A few different gateway deployment patterns are shown below. Note that more than one of these patterns can be used within the same cluster.

Shared gateway

In this model, a single centralized gateway is used by

many applications, possibly across many namespaces. Gateway(s) in the `ingress` namespace delegate ownership of routes to application namespaces, but retain control over TLS configuration.



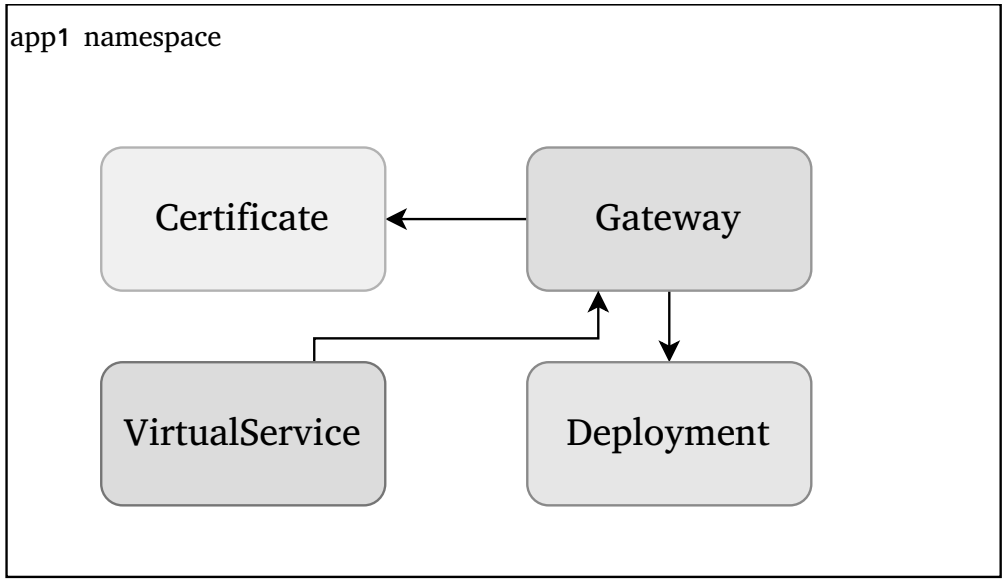
Shared gateway

This model works well when you have many applications you want to expose externally, as they are able to use shared infrastructure. It also works well in use cases that have the same domain or TLS certificates shared by many applications.

Dedicated application gateway

In this model, an application namespace has its own

dedicated gateway installation. This allows giving full control and ownership to a single namespace. This level of isolation can be helpful for critical applications that have strict performance or security requirements.



Dedicated application

Unless there is another load balancer in front of Istio, this typically means that each application will have its own IP address, which may complicate DNS configurations.

Upgrading gateways

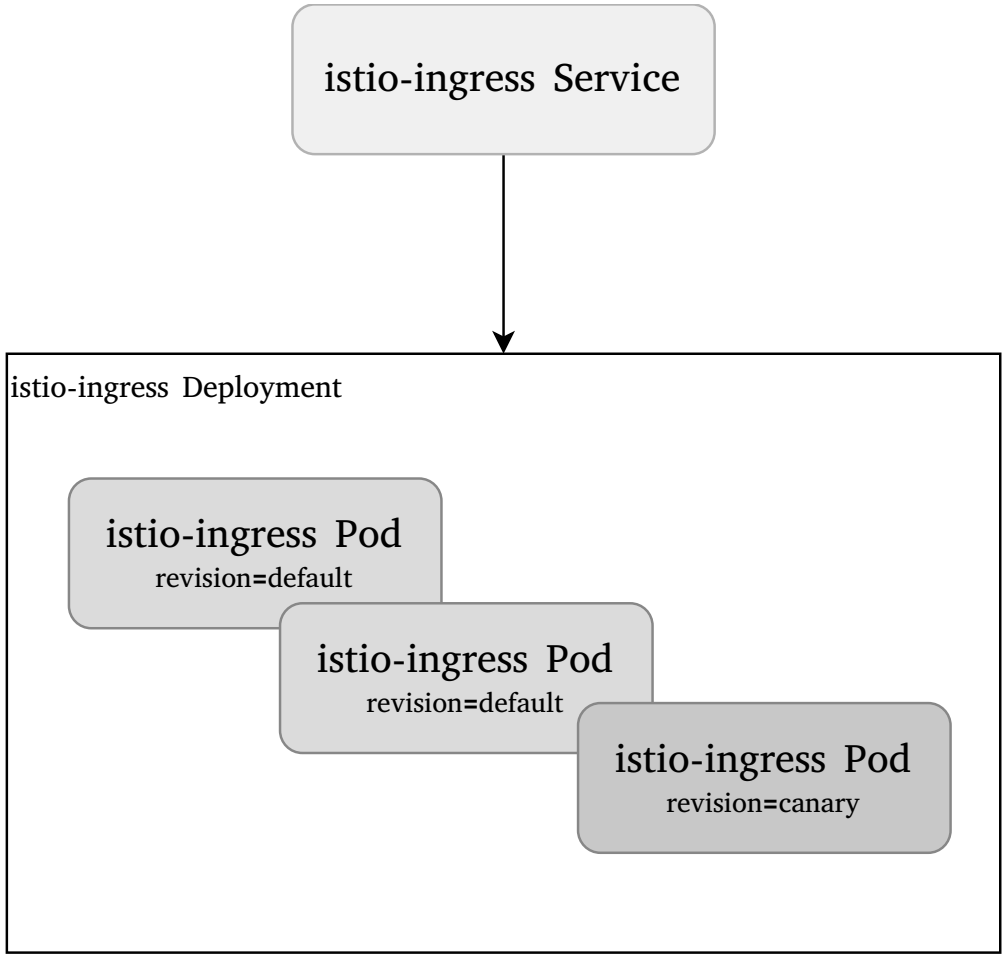
In place upgrade

Because gateways utilize pod injection, new gateway pods that are created will automatically be injected with the latest configuration, which includes the version.

To pick up changes to the gateway configuration, the pods can simply be restarted, using commands such as `kubectl rollout restart deployment`.

If you would like to change the control plane revision in use by the gateway, you can set the `istio.io/rev` label

on the gateway Deployment, which will also trigger a rolling restart.



In place upgrade in

progress

Canary upgrade (advanced)

This upgrade method depends on control plane revisions, and therefore can only be used in conjunction with control plane canary upgrade.

If you would like to more slowly control the rollout of

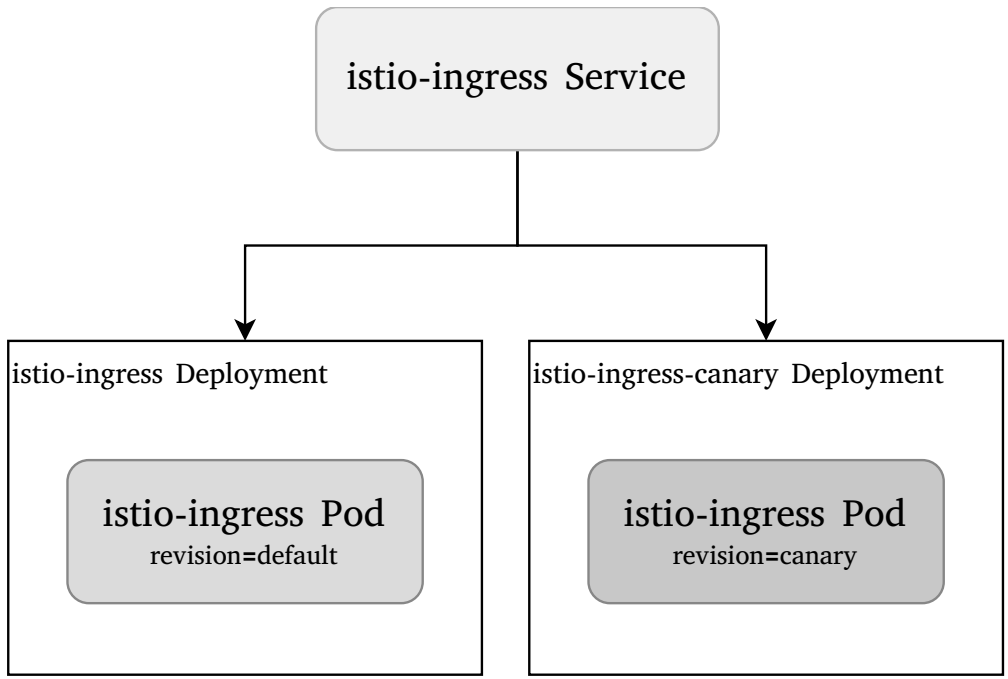
a new control plane revision, you can run multiple versions of a gateway deployment. For example, if you want to roll out a new revision, `canary`, create a copy of your gateway deployment with the `istio.io/rev=canary` label set:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-ingressgateway-canary
  namespace: istio-ingress
spec:
  selector:
    matchLabels:
      istio: ingressgateway
  template:
    metadata:
```

```
      annotations:
        inject.istio.io/templates: gateway
      labels:
        istio: ingressgateway
        istio.io/rev: canary # Set to the control plane revision
you want to deploy
    spec:
      containers:
      - name: istio-proxy
        image: auto
```

When this deployment is created, you will then have two versions of the gateway, both selected by the same Service:

```
$ kubectl get endpoints -o "custom-columns=NAME:.metadata.name,P
ODS:.subsets[*].addresses[*].targetRef.name"
NAME                    PODS
istio-ingressgateway    istio-ingressgateway-788854c955-8gv96,ist
io-ingressgateway-canary-b78944cbd-mq2qf
```



Canary upgrade in progress

Unlike application services deployed inside the mesh, you cannot use Istio traffic shifting to distribute the traffic between the gateway versions because their traffic is coming directly from external clients that Istio does not control. Instead, you can control the distribution of traffic by the number of replicas of each deployment. If you use another load balancer in front of Istio, you may also use that to control the traffic distribution.

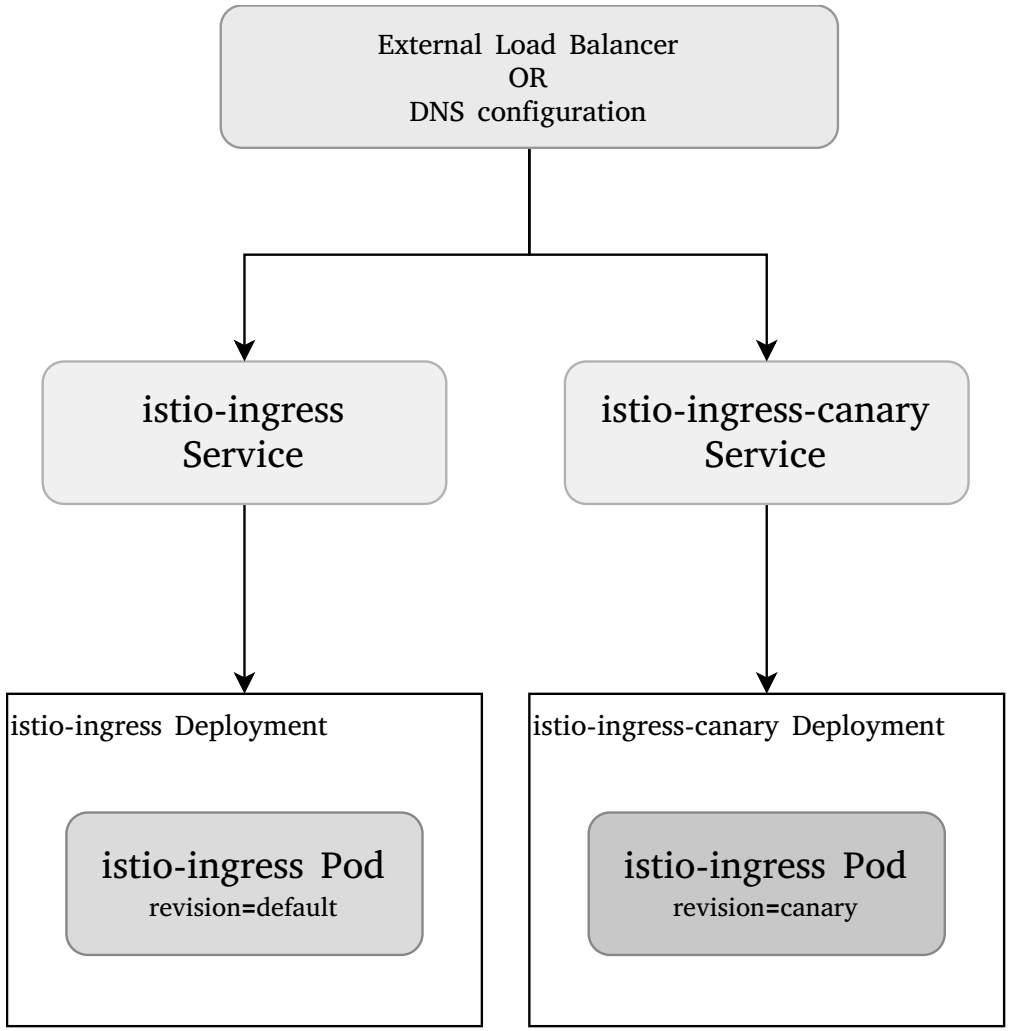
> Because other installation methods bundle the gateway `Service`, which controls its

external IP address, with the gateway `Deployment`, only the `Kubernetes YAML` method is supported for this upgrade method.

Canary upgrade with external traffic shifting (advanced)

A variant of the `canary upgrade` approach is to shift the traffic between the versions using a high level

construct outside Istio, such as an external load balancer or DNS.



Canary upgrade in progress with external traffic shifting

This offers fine-grained control, but may be unsuitable or overly complicated to set up in some environments.